Appl Intell (2018) 48:687–702
DOI 10.1007/s10489-017-1003-3

Bio-inspired metaheuristics: evolving and prioritizing software test data

Mukesh Mann[1] · **Pradeep Tomar**[1] · **Om Prakash Sangwan**[2]

Published online: 26 July 2017
© Springer Science+Business Media, LLC 2017

Abstract Software testing is both a time and resource-consuming activity in software development. The most difficult parts of software testing are the generation and prioritization of test data. Principally these two parts are performed manually. Hence introducing an automation approach will significantly reduce the total cost incurred in the software development lifecycle. A number of automatic test case generation (ATCG) and prioritization approaches have been explored. In this paper, we propose two approaches: (1) a pathspecific approach for ATCG using the following metaheuristic techniques: the genetic algorithm (GA), particle swarm optimization (PSO) and artificial bee colony optimization (ABC); and (2) a test case prioritization (TCP) approach using PSO. Based on our experimental findings, we conclude that ABC outperforms the GA and PSO-based approaches for ATC.G Moreover, the results for PSO on TCP arguments demonstrate biased applicability for both small and large test suites against random, reverse and unordered prioritization schemes. Therefore, we focus on conducting a comprehensive and exhaustive study of the application of metaheuristic algorithms in solving ATCG and TCP problems in software engineering.

Keywords Automatic test case generation · Test case prioritization · Genetic algorithm · Artificial bee colony · Particle swarm optimization

1 Introduction

With the growth of software systems and their complexity, the total cost incurred by the verification and validation process increases as the defect probability increases. Defects in software can prevent it from functioning correctly or cause it to crash. Therefore, it is essential to prevent software defects; however, it is almost impossible to deliver defect-free software. Thus, it is desirable to minimize defects as much as possible, especially when considering safety-critical systems.

A study conducted by the National Institute of Standards and Technology [1] has shown that software defects are so harmful and common that they cost the U.S. economy $59.5 billion each year. Current research and approaches are still not sufficiently mature to address this subject bravely; hence, there is an immediate need for further research. To date, many methods have been proposed to validate software systems, including code inspection and review; the most common method among them is software testing. However, this method suffers from a cost issue. Research studies have shown that software testing may consume 50 percent of the total cost of software development [2]. Reducing this cost demands the automation of the testing process. Because testing has many components, the essential, costly sub-component of “test data generation and prioritization” is unfortunately performed manually with little automation

✉ Mukesh Mann
mukesh.gbu@gmail.com

Pradeep Tomar
parry.tomar@gmail.com

Om Prakash Sangwan
sangwan_op@yahoo.co.in

1 Department of Computer Science and Engineering, School of Information & Communication Technology, Gautam Buddha University, Greater Noida, Uttar- Pradesh, India

2 Department of Computer Science and Engineering, Guru Jambheshwar University of Science and Technology, Hisar, Haryana, India

success rate. Approximately 40 percent of the total software testing cost is incurred during the test data generation process. Therefore, to reduce the cost of software testing, the success of automatic test data generators is essential. They boost the confidence of development teams by increasing the reliability of software.

Because the objective of many proposed ATCG method is to reduce the extreme cost, we must incorporate excellent and efficient methods supplemented by the highest level of experienced test automation. Hence, the demand for the automatic generation of effective test cases is increasing to incorporate completeness and reduction in terms of coverage and expenses, respectively. An effective test case not only reveals faults early in the testing phase but also uncovers untouched paths and hence reduces the software cost.

The second concern in the software industry is to perform regression testing thoroughly. The three well-known aspects of regressing testing are as follows: (1) test case minimization; (2) test case selection; and (3) test case prioritization (TCP). In this paper, our second prime objective is to focus on TCP. TCP is studied to evaluate whether reordering the test case and then executing it can reveal more faults in a program [3]. Past works have shown how different algorithms/techniques [4–8] have been proposed for TCP and a number of criteria have been used to check the effectiveness of reordered test cases [3, 9, 10].

In this paper, we address the ATCG and TCP problems. To address ATCG, a path-specific test (PST) data generation approach is proposed and experiments are conducted for several benchmark problems, including a real case study. The genetic algorithm (GA), particle swarm optimization (PSO), and ABC are used as optimization algorithms to generate the test data by solving the objective fitness function as proposed using a PST-based approach. We compare the effectiveness of these three optimization techniques in terms of their time efficiency for the generation of test data. To address TCP, PSO is used and evaluated using a large, real dataset.

The remainder of the paper is organized as follows: In Section 2, we briefly discuss related work. In Section 3, we discuss the PST approach in detail. In Section 4, we discuss the experimental setup in detail. In Section 5, we discuss the results for ATCG. We dedicate Section 6 to TCP using PSO, and finally, in Section 7, we provide a brief summary and the future scope of the present work.

2 Related work

During recent years, ATCG has emerged as a great tool for software practitioners to generate test data efficiently. Several approaches have been incorporated to generate test data efficiently. However, unfortunately, very few of them are automated. In industrial settings, many ATCG approaches have demonstrated their effectiveness for simple programs. Generally, many industrial programs exhibit some common characteristics, such as (1) a large input space and (2) high complexity and widespread structural features. Thus, many automatic test data generators have demonstrated limited success in the industrial domain.

Random testing [11–14] is a historical approach to generating a large amount of test data, which may or may not exercise a test requirement. Regardless of its simplicity, random testing has not demonstrated tremendous support for many practical purposes because it may generate a sufficient amount of good test data for simple programs, but most of the time it fails for larger, complex programs [15]. By contrast, software practitioners that have a substantial understanding of the domain of complex operations in an SUT can profitably use analysis-oriented techniques [16, 17]. However, such techniques are not used in practice because, in a development team, not all members have expertise in all the operations of SUT, especially when the system domain is new.

Many early attempts at ATCG demonstrated little success [18]. PST data generators can demonstrate good performance by yielding test data. The main idea behind such generators is to design an objective function based on numerical functions, which can be maximized or minimized further to obtain solutions. These functions can be minimized or maximized using a chosen optimization technique. McMinn (2004) [19] provided a complete survey of work in this area.

The proposed PST-based approach differs from early attempts at ATCG because path-based test data generator approaches [20–24] operate in two stages: (1) the paths are selected by analyzing the SUT; and (2) test data are generated to execute the selected path. However, such generators encounter a problem with infeasible paths. The generators fail to generate input data for such paths and the quality of the generated input is random because it is not always possible to determine the path for which the input is generated. Thus, from the large amount of input data generated, only very few inputs are useful.

However, for the proposed PST approach, the paths are selected based on a number of independent paths, and to avoid the extra effort required to generate data for each individual path, random test data are first generated and the paths that have been covered by such data are checked. Any such path is excluded from the generators, and for the remaining paths, data are generated by forming an objective function that follows each incoming constraint in the paths. These constraints traverse through the selected paths' results to form an efficient fitness function. Thus, the PST-based approach ensures that we determine every selected path that was missing in earlier attempts in this domain.

Additionally, many earlier ATCG approaches encountered the problem of determining global minima. This is possibly because of an unguided search by the search algorithms in large areas of the search domain. Many search algorithms do not determine global values because of insufficient directional information to guide these algorithms. It is important to analyze the SUT because the success of any optimization algorithm depends strongly on the program's structure. Therefore, several works have demonstrated the application of various optimization algorithms in this domain. For instance, GA was used to construct the tools GADGET [25] and EVOSUITE [26] for data generation, and the approach presented by [27] used simulated annealing for ATCG. For a more detailed discussion on current research trends in ATCG, refer to the work of Malhotra and Khari [28].

The second prime focus in this paper is TCP. Various approaches have been proposed during recent years that use different techniques and criteria to prioritize given test cases [29–33]. The most popular unit framework, called Junit, is applied to coverage-based prioritization techniques [29]. The prioritized execution of JUnit test cases results in a higher value of the Average Percentage of Fault Detected (APFD) than untreated ordering. In a controlled environment, the effectiveness of the prioritization can be measured by executing the test cases according to the fault detection rate [34].

An empirical study was conducted [31] for TCP using seven C programs. The APFD is considered as a performance metric. The result of this study demonstrated that TCP improves the rate of fault detection. The authors described several important aspects of the TCP problem, for example, a TCP can be used initially or during the regression testing of software. An empirical study on TCP was conducted [35] using a greedy algorithm, additional greedy algorithm, optimal greedy algorithm, and GA. The study aimed to examine the suitability and effectiveness of search-based techniques for the TCP problem. The study demonstrated that the greedy algorithm performed well for TCP.

ABC [36] has been presented for TCP. The behavior of scout bees and forager bees has been observed. The maximum code coverage was considered as an adequacy criterion. The average percentage of conditions covered was used as a performance metric. The proposed algorithm had shortcomings, as follows. (1) In natural BCO, the number of scout bees is 5%–10% of forager bees; however, in the proposed algorithm, the number of scout bees and forager bees was equal. (2) The energy reduction rate was chosen such that only 50% of the test suite was executed, which means that the remaining test suite was unexplored. (c) The approach was not automated, and the proposed algorithm was manually executed, which may only be useful for a small problem.

A systematic study on TCP [37] concluded the following. (1) A public dataset should be given more priority over proprietary datasets in the case of TCP. (2) More studies on the comparison of prioritization methods are required. (3) Priority should be given to using industrial projects that represent real-time industrial problems.

A TCP technique was proposed [38] using the GA. The test case information available from regression testing was used during prioritization using the GA. The GA took the test case information from regression testing as an input and produced a sequence of test cases to be executed so that the maximum number of modified lines could be covered. The prioritization of test cases was performed based on a number of modified lines covered by a test case; the test case that covered the fewest modified lines was given the lowest priority and executed last, provided the deadline time was not reached. The authors evaluated their study on a small problem domain, and a real benchmark study was missing from their work.

Additionally, the works [28, 35, 36, 39] and [40] demonstrated the application of various metaheuristics in the TCP domain. The problems that were encountered by earlier studies in this domain were clearly highlighted in the works of [37, 38], which emphasized the evaluation of TCP on more real datasets, together with a comparison with earlier studies. Thus, taking into account the guidelines from [37, 38] and other works in this domain, it has been observed that proposing a metaheuristic for TCP is beneficial only if it is evaluated on large, real datasets, together with a comparison with existing techniques for TCP.

Therefore, in this paper, we focus on conducting a comprehensive study of optimization algorithms in the domain of ATCG and TCP. In Sections 3–5, we propose a PST-based methodology for ATCG, together with its results. In Section 6, we formulate the TCP and address using a PSO-based approach. Throughout the paper, our aim is to address ATCG and TCP using metaheuristic algorithms.

3 Methodology

Our methodology for path specific target-based test data generation (PST) consists of four steps, as shown in Table 1. The first phase is used to clearly specify all testing requirements that are derived based on various condition decisions/predicate nodes. A condition coverage table is then formed, which is used to mark off the conditions that have been covered and not covered in the SUT.

The second phase is the generation of a random test case, which is executed against the SUT to check which subset of conditions is covered. Thus, the initial coverage table is initialized. The last phase is the application of the metaheuristic algorithm by forming a suitable fitness function

Table 1 Automatic test case generation process

Step 1:	Initialization Step: Analysis of SUT
	• Drive testing requirements according to condition/predicate decision in SUT.
	• Prepare condition-decision test metric table.
Step 2:	Initialization of test metric table
	• Randomly generate a single test case.
	• Execute generate test case against SUT.
	• Monitor the condition coverage information using this test case.
	• Initialize coverage table with the obtained information.
Step 3:	Generate test cases to fulfill all test requirement in test metric table using search process (using Metaheuristic optimization algorithms)
	• Generate test case by applying selected evolutionary algorithm on the selected test requirement from test metric table.
	• Record the test cases, coverage information, and other relevant information to update the test metric table.
Step 4:	Repeat step 3 until all test requirements in test metric table are met.

using PST on each test requirement in the coverage table that is not covered. The metaheuristic algorithm is repeatedly applied until we obtain a fully tested coverage table.

A test's metric is the adequacy criterion that tracks which conditions are covered by the decision taken after their execution. For example, consider the following code segment:

If$(a + b > c \&\& b + c > a \&\& c + a > b)$
{
Prints "true";
}
else {
Prints "false";
}

To execute this condition, a tester must obtain an input value (test case). With this value, the code segment can take two decisions: true or false.Thus, every test case must have the outcome of either true or false at least once. However, it should be noted that entire decision for a particular condition depends on all sub-conditions that constitute the full decision condition. For example, consider the condition If$(x < 0$ *and* $y > 1)$; it consists of two sub-conditions: $(x < 0)$ and $(y > 1)$.

The test case shown in Table 2 ensures that each sub-condition has a true or false outcome at least once.

Thus, none of the four conditions had the outcome "true." To summarize, the outcome of the full condition depends on the sub-conditions and the operators joining these sub-conditions. Thus such condition and its decisions are more complicated and reliable than branch and statement coverage. This motivates us to consider decision conditions as a test metric criterion in the present work.

To reach a particular condition in an SUT, we first need to reach the sub-conditions that follow, which ensures that we reach the goal condition. Thus, there must be an approach to arrive at all sub-conditions.

A particular goal condition can consist of many different sub-conditions, and to reach the goal, we must follow the sub-conditions such that their execution takes the flow of the program to the goal state.

For example, consider the triangle classification code segment given below:

1: if (tri == 1) System.out.println ("Scalene triangle");
2: else if (tri == 2) System.***out***.println ("isosceles triangle");
3: else if (tri == 3) System.***out***.println ("equilateral triangle");
4: else if (tri == 4) System.***out***.println ("not a triangle");

Table 2 Test cases for sub-conditions

X	Y	OUTCOME
−2→ true	0→false	False
0 → false	0 →false	False
5 → false	5 → true	False
1 →false	−1 →false	False

Table 3 Example of a test metric

Line no	True	False
1	–	X
2	–	X
3	X	–
4	–	–

Table 4 Fitness function for predicate condition decision expressions

Expression	a == b	a! = b	a<b	a<=b	a>b	a>=b	a||b	a&&b
False	−|a−b|	abs(a−b)	b−a	b−a	a−b	a−b	f(c1)+f(c2)	min(f(c1), f(c2))
True	|a−b|	−abs(a−b)	a−b	a−b	b−a	b−a	min(f(c1),f(c2))	f(c1)+f(c2)

Table 3 illustrates the test metric, which specifies that for the decision status for each condition with a supplied test case (5, 5, 5), it prints "equilateral triangle." The test case covered the false, false, true branches of statements 1, 2 and 3, respectively. The tester can easily track which true/false condition for a particular branch has been covered or not covered. Based on Table 3, testers can apply the metaheuristic optimization algorithm proposed in this paper to generate a test case that can exercise the false, false, and true conditions of statements 1, 2 and 3, respectively.

3.1 Optimization objectives

Dynamic test case generation is based on the fact that a well-formulated problem that can determine a test case can be transformed into a numerical maximization or minimization problem. For every such problem, we can form a function that can be maximized or minimized using different available optimization techniques. This function must guide the search process for test case generation that can satisfy a given condition. For every branch/predicate node in the SUT, the function is called an objective function. An objective function can evaluate the goodness of the generated test case, that is, how good the generated test case is in satisfying the particular condition for which it is generated.

If there are conditions that are composed of several other sub-conditions and connected using "*" (AND) or "||" (OR) operators, then the following rule is followed to form the combined fitness for the entire expression:

IF cond "a" AND cond "b" AND cond "c."

The objective fitness for the entire expression that results in a true branch is

$$¥_{and} = ¥_{1a} + ¥_{1b} + ¥_{1c},$$

where ¥is the objective function for the entire expression, and $¥_a$, $¥_b$, and $¥_c$ are the objective functions for cond "a," cond "b," and cond "c," respectively.

Similarly, if the expression is *IF cond "a" OR cond "b" OR cond "c,"* then the objective fitness for the entire expression that results in a true branch is

$$¥_{or} = \min\ (¥_{1a,} ¥_{1b}, ¥_{1c}).$$

Table 4 provides various fitness functions that can be used to evaluate the true and false branches of an expression [41].

We consider a small fragment of code to understand the construction of the objective fitness function that can be minimized or maximized using the proposed metaheuristic approaches:

```
If (a + b > c&&b + c > a&&c + a > b)
{
...................
}
```

To reach the true condition of the decision, If (a+b>c && b+c>a && c+a>b), there must be the following sub-conditions, which must be true to exercise the truthiness of the decision:

1. a+b>c→ must be true, therefore the objective function ($¥_1$ must be $¥_1 = (c-(a+b))$.
2. b+c>a → must be true, therefore the objective function ($¥_2$) must be $¥_2 = (a-(b+c))$.
3. c+a>b → must be true, therefore the objective function ($¥_2$) must be $¥_2 = (b-(c+a))$.

Because the decision consists of three sub-condition decisions, the net objective function for the decision is $¥= ¥_1+¥_2+¥_3$.

In a similar manner, we can form objective functions for different test requirements that arise while traversing a path.

3.2 Complete illustration

We illustrate the approach by considering Tri_tryp [42–47] as a sample benchmark problem. The test requirement is to generate a test case for an *equilateral triangle* (***Tri_Tr1***).

Fitness function construction We individually derive the objective fitness function for each targeted goal by constructing fitness functions as explained above (see Table 4). For example, to reach the goal *equilateral triangle* (*Tri_Tr1*), the following sub-goals (sub-conditions), together with their decisions, must be followed to reach a final objective function that guarantees the generation of a test case that exercises the targeted goal. Table 5 shows the fitness function for each sub-condition in the path to the final goal: Tri_Tr1.

Thus, we can form the net PST-based fitness function for any targeted path. An instrumented MATLAB version of triangle classification is shown in Table 6.

Table 5 Fitness for a targeted goal (Tri_Tr1)

Sub goals/test requirements	Condition	Fitness value
If (a+b>c && b+c>a && c+a>b)	True	¥1=(c− (a+b))+ (a−(b+c)) + (b−(c+a))
If((a!=b&&b!=c&&c!=a))	False	¥2=min(min(abs(a−b),abs(b−c)), abs(c−a));
Else if (((a==b) && b==c))	True	¥3=abs(a−b)+ abs(b−c)
Net fitness(¥)	¥1+¥2+¥3;	

Thus, the source code fitness function can be created for targets. With the formed fitness function, the search process is conducted using optimization search algorithms. We investigate whether the designed fitness function is sufficiently good to generate the target values when used by the optimization algorithms in this study. The initial solution space of each search algorithm in this study is assigned based on the problem bounds. For example, in the case of Tri_tryp, there are three variables in the range for the lower and upper bounds, [1 1 1] and [100 100 100], and each bound represents a potential solution to the problem. The objective is to reach to an optimal solution by evaluating the formed fitness function using the metaheuristics in this study. By *optimal,* we mean that the generated value is sufficiently fit to traverse the targets for which it is generated. Below is a sample approach to instrument different optimization algorithms using the fitness function formed using the PST-based approach:

```
% calling type of metaheuristic
Call = @ GA/PSO/ABC[1]
% setting max run
Total run = 20;
% call to fitness function formed using PST based approach.
FitnessFunction = @objective_funtion; % In case of Tri_Typ the fitness function is Tri_classification
% setting number of variables in problem
NumberofVariables = []; % set according to problem domain
% set lower and upper bound of variables
lb = [];
ub = [];
% setting other parameters as per Algorithms setting (refer Section 4)
print ('Final Solution obtained', Fval);
print ('Total Time taken');
```

Each subject in this study is evaluated for their test requirements. The solutions obtained by evaluating the fitness function using metaheuristics are then compared. In the next section, we discuss the basic implementation decisions and parameter settings used for different metaheuristics.

4 Experimental setup and optimization algorithm details

Three search-based metaheuristics were used in this study: GA, PSO, and ABC. MATLAB 7.10 was used for the implementation. In the following, we describe the basic implementation decision for each metaheuristic used in this study.

1. *GA:* A double vector population was used as an initial population type. The selection scheme was a roulette wheel with a uniform creation function to create a uniformly distributed initial range in two spaces for the lower and upper bounds. Each space of the lower and upper bounds was selected based on the problem bounds. A crossover fraction of 0.9 with a single crossover was used for the crossover phenomena. The initial population size was set equal to 100.
2. *PSO:* The initial population was created uniformly between the lower and upper bounds of the problem. The inertial maximum and minimum weights, and acceleration factor were all set to one. A maximum of 100 iterations was set to run the experiment.
3. *ABC:* The colony size was set to 100 and initial food sources were created uniformly between the lower and upper bounds of the problem, as in the case of PSO. The total food sources was set to half the colony size. A maximum of 100 iterations was set to run the experiment.

4.1 System under test

In an ideal situation, we could select a large number of programs for the experiments, which might provide a good representation of a large experimental space. Unfortunately, this ideal assumption is hypothetical for many reasons: first, there are no universal sampling criteria available to select a sampling size from the large universe of software programs.

[1] Detailed MATLAB code can be provided by the authors upon prior request for academic use only.

Table 6 Instrumented Tri_Tryp program

Function [type, Line_no]= Tri_classification(x,y,z)

% net Fitness function for first independent path, Scalene triangle $f_{Scal} = f_{0t}+f_{1t}+ f_{2t}$

% net Fitness function for second independent path, equilateral triangle $f_{equi} = f_{0t}+f_{1t}+ f_{2t}+f_{3t}$

% net Fitness function for first independent path, isoceleous triangle $f_{iso} = f_{0t}+f_{1t}+ f_{2t}+f_{3f}+ f_{4t}$

% net Fitness function for first independent path, invalid triangle is $f_{inv} = f_{0t}+f_{1t}+ f_{2f}+ f_{3f}+ f_{4f}$

% where subscript in fitness function f shows (1) first subscript as line number (2) second subscript as either false(f) or true(t) to represent that this function is for false branch, *for example,* a fitness function f_{1t} means that this fitness function is formed for the True Branch of line no_1

% f_{0t} =(0−x)+(0−y)+(0−z)

% f_{0f} =min((min(x−0),(y−0)),(z−0))

1. if(x>0 && y>0 && z>0)

% f_{1t} =(z− (x+y))+(x−(y+z)) +(y−(z+x))

% f_{1f} =((x+y)−z)+((y+z)−x) +((z+x)-y)

2. if (x+y>z && y+z>x && z+x>y)

%f_{2t} =−abs(x−y)+−abs(y−z)+−abs(z−x));

% f_{2f} =min(min(abs(x−y),abs(y−z)), abs(z−x));

3. if(x∼=y&&y∼=z&&z∼=x)

4. type= 1; % scalene

5. L=4;

% f_{3f} =abs(x−y)+abs(y−z)+abs(z−x);

% f_{3t} =min(min(−abs(x−y),−abs(y−z)),−abs(z−x));

6. elseif ((x==y) && (y==z) && (z==x))

7. type= 2; % Equilateral

8. L=7;

% a1=(abs(x−y)+−abs(y−z));

%a2=(abs(y−z)+abs(z−x));

%a3=(abs(z−x)+abs(x−y));

%b1=min(-abs(x-y),abs(y−z));

%b2=min(abs(y−z),abs(z−x));

%b3=min(abs(z−x)+abs(x−y));

%f_{4t} =min(min(a1,a2),a3);

%f_{4f} =b1+b2+b3;

9.elseif (((x==y) && (x∼=z)) || ((x == z) && (x∼= y)) || ((y == z) && (y ∼= x)))

10. type= 3; % Isoceleous

11. L=10;

12. else

13. type= 4; % invalid triangle

14. L=3;

15. end

16. else

17. type= 5; % out of range

18. L=2;

19. end

20. Line=L;

21. tri=type;

22. end

Therefore, any chosen sampling criteria are always ad hoc in nature and a matter of open debate. Second, many works on optimization algorithms for target-oriented test data generation have demonstrated its application on a small set

of programs. If we select programs that are beyond the limits of currently available techniques, such as GA, then the present study will not accumulate any relevant information.

Thus, we considered the aforementioned factors and selected ten benchmark problems. In addition to the traditional research benchmark problems, we also considered validating the proposed work on a larger, real application. Hence, we considered a real case study of a popular e-commerce website: *Flipkart* [48]. A module to purchase a mobile gadget was considered by writing the source code of the data flow functionality of this module. In the following are the benchmarks considered for the ATCG study:

1. *Bubble sort*: a sorting program that recursively sorts a list, compare adjacent items, and swaps them into the correct order. This program has two test requirements.[2]
2. *Leap year*: checks whether a year entered by the user is a leap year. This program has two test requirements.
3. *Factorial:* calculates the factorial of a given number as its output. This program has one test requirement.
4. *Greater of three numbers:* checks the greater number from the input three numbers. This program has four test requirements.
5. *HCF*: calculates the highest common factor (HCF) of two input numbers. This program has four test requirements.
6. *Quadratic equation*: determines the roots of a quadratic equation as real or imaginary numbers when the coefficients are given as an input. This program has three test requirements.
7. *Student grade*: determines the grade of the student when the input arguments are the subject's marks. This program has four test requirements.
8. *Sum and average*: determines the summation and average of input integers; the size of the input array can be n. This program has one test requirement.
9. *Triangle classification*: classifies a triangle as "scalene," "equilateral," "isosceles," and "not a triangle" based on the input three integers as three sides of the triangle. This program has three test requirements.
10. *Flipkart ecommerce application*: source code mapped from the functional flow of the e-commerce application "*Flipkart*" [48]. If a user wants to buy the electronic item "mobile," then he/she should traverse the functionalities as follows: *Home→gadgets→mobile→prize range→select mobile→select quantity→checkout (bill)*. These functionalities are mapped into source code and different test requirements are tested. For example, a test requirement is to buy an Apple mobile in the range

Table 7 Relative performance of metaheuristic algorithms for ATCG

Subjects	Subject's Name	Total Path Test Requirements	Bounds		Average time (Sec) in 20 runs			% time saving in ABC	
			Lower Bound	Upper Bound	GA Time/S.D	PSO Time/S.D	ABC Time/S.D	*w.r.t* GA	*w.r.t* PSO
S1	Bubble sort	2	[1 1 1 1 1]	[100 100 100 100100]	2.22/0.0036	0.36/0.0032	0.095/0.0003	95.72	73.61
S2	Leap year	2	[2000]	[4000]	1.16/0.0028	0.20/0.0024	0.092/0.0002	92.06	54
S3	Factorial of a number	1	[1]	[5]	0.95/0.0026	0.099/0.0005	0.046/0.0002	95.15	53.53
S4	Greater of three numbers	4	[1 1 1]	[100 100 100]	3.68/0.0024	0.38/0.0023	0.16/0.0028	95.65	57.89
S5	HCF	4	[1 1]	[5 5]	3.61/0.0027	1.88/0.0022	0.15/0.0025	95.84	92.02
S6	Quadratic equation	3	[1 1 1]	[100 100 100]	3.69/0.0029	0.51/0.0030	0.14/0.0027	96.20	72.54
S7	Student grade	4	[1 1 1]	[100 100 100]	2.36/0.0031	0.28/0.0031	0.13/0.0027	94.49	53.57
S8	Sum and average	1	[1 1 1 1 1]	[5 5 5 5 5]	0.62/0.0024	0.076/0.0002	0.051/0.0002	91.77	32.89
S9	Triangle classification	3	[1 1 1]	[100 100 100]	2.04/0.0026	0.21/0.0028	0.15/0.0024	92.64	28.57
S10	Flipcart ecommerce application	135	[1 3 5000 11 1]	[3 10 10000 21 10];	23.62/0.0030	11.07/0.0028	4.99/0.0034	78.87	54.92

[2] A test requirement identifies what needs to be tested, including conditions, business logic, and functional and non-functional benchmarks.

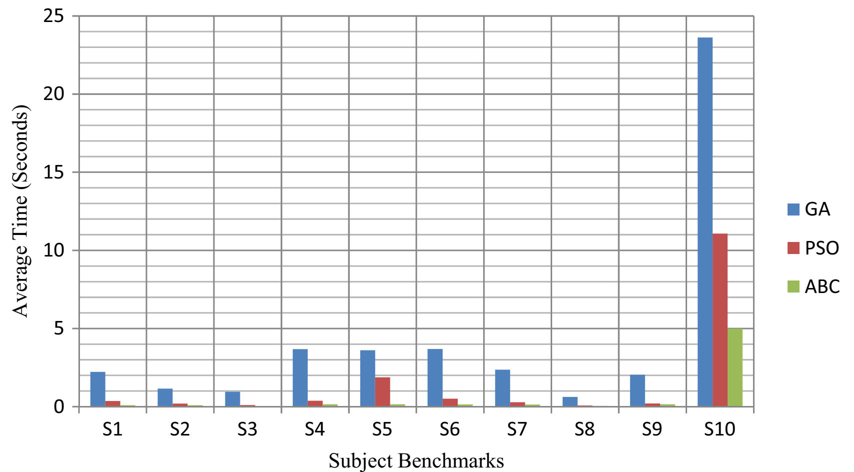

Fig. 1 Relative performance of GA, PSO, and ABC for ATCG

$800–$1000 when the quantity is two. The tester has to check whether the final item in the checkout is in accordance with the specification. For this, the source code is traversed to fit the test requirement, and the final objective function is created, which is optimized using the proposed metaheuristics. This program has 135 test requirements.

5 Results and discussion

Each subject benchmark was taken to read their test requirements, and the corresponding objective functions were created. For each test requirement, the corresponding objective function was minimized using the GA, PSO, and ABC metaheuristics. Test data was generated using all three metaheuristics. We aimed to determine which search algorithm was more efficient for ATCG. For performance measures, we considered two criteria: (1) time taken to generate the final test value; and (2) coverage provided by the generated test case.

Table 7 shows the relative performance of PSO, GA, and ABC in terms of the average time taken for 20 runs to generate the final test data for each subject program.

To establish confidence in the proposed approach, a statistical test was performed to evaluate whether ABC was significantly better than GA and PSO for ATCG on selected artifact instances. As discussed previously, 10 instances of different benchmark problems, including a real benchmark instance, were chosen to run each algorithm. Each algorithm was run on 10 subjects, n times, where n=20, and the average value from n runs was collected for each subject problem. This constituted a total of $3\times10\times20 = 600$ runs. Additionally, each algorithm was evaluated based on the final time taken to generate a test case that could traverse each targeted goal for a given problem. The average values of time obtained are as follows: ABC (0.095, 0.092, 0.046, 0.16, 0.15, 0.14, 0.13, 0.051, 0.15, and 4.99), GA (2.22, 1.16, 0.95, 3.68, 3.61, 3.69, 2.36, 0.62, 2.04, and 23.62), and PSO (0.36, 0.2, 0.099, 0.38, 1.88, 0.51, 0.28, 0.076, 0.21, and 11.07). We sought to understand and evaluate whether there was any significant difference between two algorithms, that is, ABC vs. GA and ABC vs. PSO.The Mann–Whitney U-test (an unpaired test at 5% level of significance) yielded a $p-$value of 0.0022 for ABC vs. GA and 0.03156 for ABC vs. PSO. For each case, the $p-$value suggested that there was a statistical difference between ABC vs. GA and ABC vs. PSO. However, it is extremely important to run a paired test, such as the Wilcoxon rank sum, when a case study involves artifacts of different levels of

Table 8 Small test case fault matrix

Test case number	Revealed fault(s)
t1=	f1
t2=	f1
t3=	f1
t4=	f1+f2
t5=	f1
t6=	f1+f3+f4+f5
t7=	f1+f2
t8=	f1
t9=	f1
t10=	f1
t11=	f1
t12=	f1
t13=	f1
t14=	f1
t15=	f1
t16=	f1
t17=	f1
t18=	f1+f2

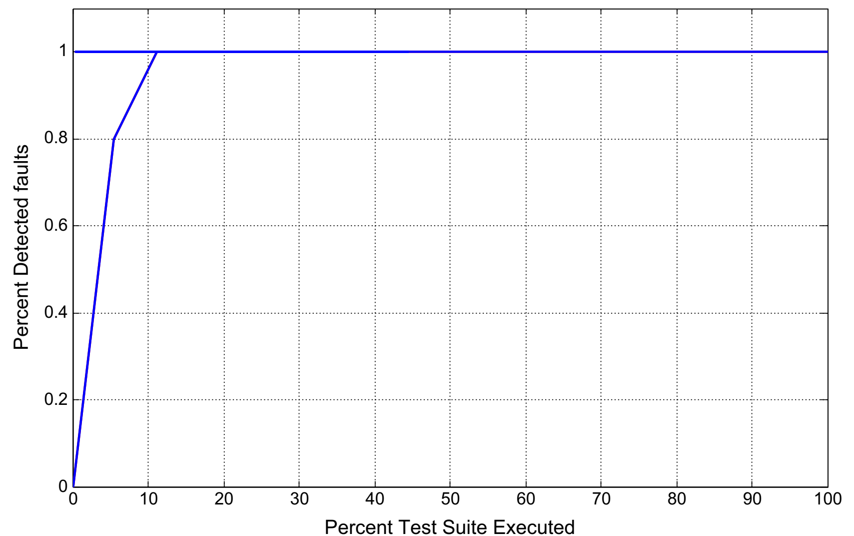

Fig. 2 Percentage of test suite executed vs. percentage of detected faults for a small application

difficulty [49]. In a paired sum test, the null hypothesis is z=0, that is, the difference $Z_i = Y_i - X_i$ is centered on zero, which means that there is no statistical difference between two algorithms, whereas the alternative hypothesis states that there is a statistical difference between two algorithms. In this study, $Z_{ABCVS.GA} = -(2.125, 1.068, 0.904, 3.52, 3.46, 3.55, 2.23, 0.569, 1.89, 18.63)$ and $Z_{ABCVS.PSO} = -(0.265, 0.108, 0.053, 0.22, 1.73, 0.37, 0.15, 0.025, 0.06, 6.08)$. On average, the first algorithm, ABC, was always better than the second algorithm in each case. A Wilcoxon rank sum test (5% level of significance; W_critical=8) in this study yielded w=0, which argues that we should reject the null hypothesis (because W<W_critical) in both cases, that is, ABC vs. GA and ABC vs. PSO. Thus, there is a significant difference in the performance of ABC vs. GA and ABC vs. PSO. Additionally, the results are not in sharp contrast with those of the aforementioned Mann–Whitney U-test. The reason to run such a paired test is to establish confidence in evaluating the goodness of the randomized algorithm when it is run over artifacts that have different difficulty levels [49].

Figure 1 shows a comparison of the efficiency of three search metaheuristics when time is considered as a performance parameter.

The standard deviation of the algorithm's time clearly indicates very few diversions from the mean time. For each subject, ABC outperformed GA and PSO.

5.1 Comments on coverage information as a performance parameter

Coverage information is considered as a good criterion to measure the potential of a test case. The coverage itself can be divided into statement coverage, branch coverage, and path coverage. Among these three types of coverage, path coverage has the most potential because 100% path coverage reflects 100% statement and branch coverage.

Because a given program can contain a number of independent paths, we considered path selection based on independent paths as a primary requirement. For each selected path, we designed a path fitness function, which was further given as an argument to the three metaheuristics,

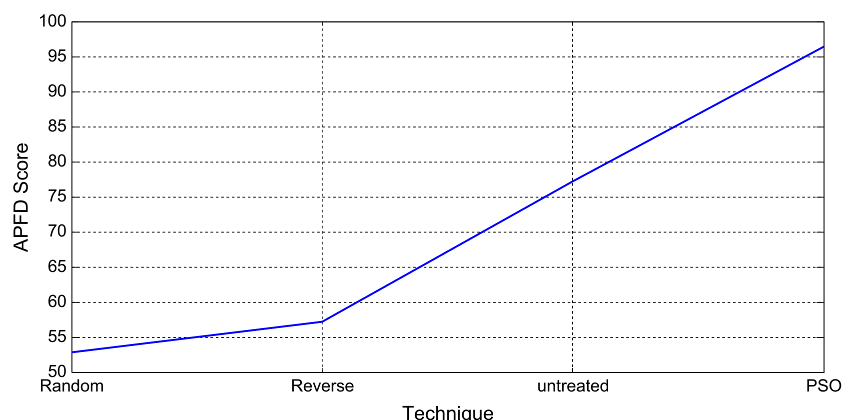

Fig. 3 APFD comparison of different prioritization schemes for a small application

Table 9 Subject benchmark real applications

S.no	Application	Number of LOC	No. of classes	No. of methods	No. of branches	No. of test cases	No. of unique faulty line
1	Tera_Paint3	18376	219	644	1277	424	148
2	Tera_Present3	44591	230	1644	3099	32	139
3	Tera_Spreadsheet3	12791	125	579	1521	1172	121

as discussed above. From the start, our goal was to hit the targets, and each final generated test data value was found to reach the goal, thereby achieving full coverage for the targeted path. Therefore, this approach can achieve 100% path coverage if the fitness function is designed efficiently.

When same number of initial maximum iterations was set, the only difference we observed was in the time taken to obtain the final values for each of the three metaheuristic algorithms. From small to large applications, the time benefits of ABC compared with GA and PSO bias its application for ATCG.

6 Test case prioritization: a swarm approach

When a program is modified or updated, it is almost impossible to guarantee that the changes work correctly and that the unmodified modules of the program have not been affected by the modifications. However, it is extremely necessary to perform exhaustive regression testing because a small change in one module can reflect a bigger change in another module. Thus, regression testing has emerged as a great subject of research within the software testing community. Three core areas of regression testing that attract most researchers are as follows: (1) test case minimization; (2) test case selection; and (3) TCP.

The second objective of this paper is to develop a TCP strategy using swarm behavior. TCP [50] is a well-studied research problem in software testing.

TCP is a technique of ordering given test cases based on defined prior criteria such that the test cases that determine a number of defects and faults are given higher priority; that is, they are executed before the others. Then the tester has an opportunity to prioritize test cases based on some criteria, such as maximum code coverage and maximum defect/fault coverage in the minimum test suite execution time. Thus, testers can save precious time and costs during regression testing. More formally, the prioritization problem is defined as follows [50]:

Definition (Test case prioritization problem) *Given: test suite T, set of permutations of T called PT, and function from PT to real numbers* $f : PT \rightarrow R$.

Problem: Determine $T' \in PT$ *such that* $(\forall \left(T''\right) \left(T'' \in PT\right) T'' \neq T') \left[F\left(T'\right) \geq f\left(T''\right)\right]$.

Ideally, function f should map from tests to their fault detection capability. However, we know the defect-finding capability of a test only after its execution. In practice, function f is taken, which can substitute fault detection capability of tests.

Our second focus is on TCP, which is used to identify the best order in which given test cases under some test suite must be executed to maximize the rate of fault

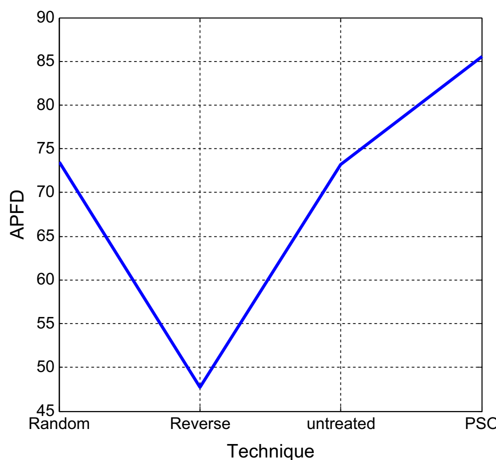

Fig. 4 APFD score for Tera_Paint

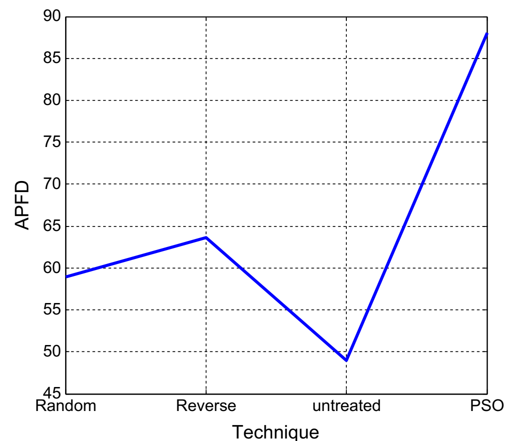

Fig. 5 APFD score for Tera_Present

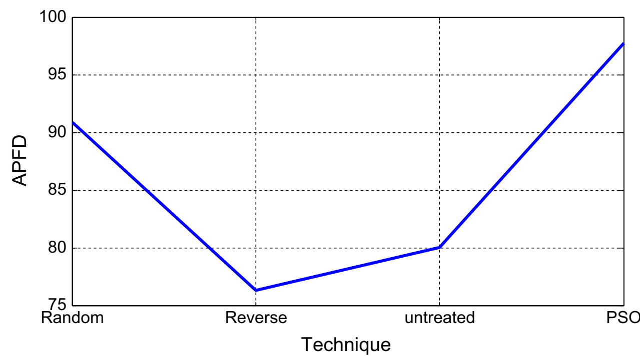

Fig. 6 APFD score for Tera_Spreadsheet

detection. Different artificial intelligence techniques are available to solve optimization problems [51–54, 56], as discussed in Section 2. To evaluate the performance of various TCP schemes, the APFD was considered as a performance parameter. A higher preference was given to the prioritization scheme that had a higher APFD value. The APFD [10] is defined as

$$APFD = [1 - \Sigma_{i=1}^{g} reveal(i, T)/ng] + 1/2n, \quad (1)$$

where T is the test suite, g is the number of faults in the program under test, n is the number of test cases, and *reveal (i, T)* is the position of the first test in T that exposes fault i.

The APFD score of the proposed swarm-based algorithm was compared with existing work using three real test case fault matrices [57–59]. For each test suite, we compared the proposed PSO prioritization scheme with the existing techniques for TCP.

Throughout the paper we are interested in the following research question:

Q) Which techniques are most effective for solving the TCP problem in software testing?

6.1 Motivational example: small test case fault matrix

To understand the PSO approach for TCP, we considered a small motivational example: *TeraPaint3*. The original repository [57] had 424 test cases with their corresponding revealed faulty line numbers. Each similar faulty line could be treated as a fault, that is, if test case T1 revealed a fault in line 120 of the application, then test case T1 revealed a fault. Because the repository of the test case fault matrix contained many test cases that revealed the same faults, to select a test case to create a small test fault matrix, we determined a fault that was captured by the maximum number of test cases. Additionally, then we ensured the compulsory selection of those test cases that at least captured the most common fault. A selected test case could capture faults other than the most common. Thus, we ensured a fair selection to create a small test fault matrix, which at least revealed the most common fault. In this example, the original test fault matrix size was 424×148, that is, 424 test cases revealed 148 faults in the application. The most common fault was on line 110. To map the fault line number to a fault number, we first identified all unique faulty lines in the program, and for each unique faulty line, a fault number was assigned. For example, let a program contain n unique faulty line numbers. To assign a fault number for each unique faulty line, we first extracted all unique fault lines in a matrix, and for each faulty line, an index starting from one up to the last unique fault was assigned. Thus, we could use this fault index instead of faulty line numbers in our test fault matrix. Hence, if a faulty line number was 110 (revealed by test case 5) and its corresponding fault index was "39," then test case 5 revealed fault 39.

To understand the working strategy, we considered a small test case fault matrix of size 18×5 from the larger *TeraPaint3* application of size 424×148. Although in a later section, we consider the original *TeraPaint3* application test case fault matrix, to clearly understand the basic working principle, we chose the 18×5 test case fault matrix (see Appendix 1). This matrix was not selected randomly; extreme care was taken to select test cases that revealed the most common faults. Thus, 18 test cases were selected that revealed the five most common faults in the original *TeraPaint3* application.

6.2 PSO objective function creation

Our next objective was to prioritize the selected small motivational test suite in order of maximum fault coverage. To apply PSO to this problem, we first created an objective function.

Table 10 Comparison of the APFD scores for various ordering in TCP

S.no	Application	APFD Score for different TCP Schemes			
		Random	Reverse	Untreated	PSO
1	Tera_Paint	73.48	47.74	73.15	85.52
2	Tera_Present	58.89	63.61	48.99	88.07
3	Tera_Spreadsheet	90.87	76.32	80.04	97.75

Because our objective was to maximize the rate of fault detection in a given test fault matrix, the objective function had to be designed in such a way that it would capture the most common fault. Considering this aim, the following objective function was formulated:

$$Netfitness = \sum_{i=1}^{n} t, \quad (2)$$

where t is the test case and t = $\sum$revealed faults.

Additionally, $n$is the number of test cases in the test suite.

Equation (2) ensured that a particular test case captured its entire corresponding faults. The initial swarm population was created randomly between the lower and upper bounds, where the lower bound was the first test case and the upper bound was the final test case in the test case fault matrix. In the present example, the size of the small test case fault matrix is [18×5]. Table 8 presents the faults revealed by each test case.

Note that

$t_6 = f_1 + f_3 + f_4 + f5$ in Table 8 indicates that test case t6 captures fault numbers 1, 2, 4 and 5.

The following fitness function was formed:

Objval = (t1 + t2 + t3 + t4 + t5 + t6 + t7 + t8 + t9 + t10 + t11 + t12 + t13 + t14 + t15 + t16 + t17 + t18).

For each iteration, the Objval fitness function was evaluated and checked for whether the population was converging toward the global maximum. The global maximum for this particular example was an arrangement of test cases that would reveal the most common faults. Thus, if the algorithm generated such a global arrangement, then it was stopped. The maximum number of iterations was 500, which was used as a stopping condition.

The following is the final output of the PSO generated test sequence:

t6 t4 t7 t18 t1 t2 t3 t5 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17.

The APFD score [3] was calculated by determining the area under the curve of Fig. 2. The APFD score calculated using (1) resulted in the same score as that calculated using the area under the curve method. To explain the results, both techniques are presented in this paper (Fig. 2). We compared the PSO TCP scheme with random, reverse, and no ordering, as shown in Fig. 3.

6.3 From motivational to real applications

This motivational example was further extended to check whether the PSO scheme could prioritize a large amount of test data. Our motivation to extend this study was based on the previous studies in TCP. A number of researchers have used small datasets in their studies on TCP [39, 40, 60–63]. Extending the study to larger datasets develops more confidence in the proposed work. Furthermore, real-time testers have to manage large amounts of data, thus a bias toward using a large test dataset is naturally justified.

Three real test case fault matrix datasets, as discussed above, were used to further check the validity of PSO-based ordering. The three real applications [57–59] are described in Table 9. The objective function for each benchmark was formed (see Appendix 2) and evaluated using the PSO algorithm.[3]

6.3.1 Pre-processing repository data

The size of the original dataset (Tera_Paint) was (424×148), where 424 is the number of test cases and 148 is the number of unique faulty lines in the application. To use this test data, we first instrument it using the following steps:

1. Assign each unique faulty line an index value, starting from one.
2. Create a fault matrix for each corresponding faulty line.
3. Create a matrix of the test case and its corresponding fault numbers.

The above three steps were also instrumented for the remaining two benchmarks, that is, Tera_Present and Tera_Spreadsheet. Thus, we obtained a matrix for each of the three applications. The matrix contained information about each test case's fault-finding capacity. The APFD scores obtained for Tera_Paint, Tera_Present, and Tera_Spreadsheet after applying PSO TCP to the formed test fault matrix are shown in Figs. 4, 5 and 6, respectively.

A comparative analysis of the different prioritization schemes is shown in Table 10.

From the above observation, it is clear that for both small and large test suite prioritization, the effectiveness of PSO-based ordering was significantly better than that of random, reverse, and unordered prioritization schemes.

7 Conclusion

In this paper, we focused on two objectives: the generation of test cases using metaheuristic approaches and TCP of large repository data using PSO. For the generation of test cases, we considered 10 software engineering benchmark problems, including one real case study, to validate the proposed approach. Three optimization algorithms were considered to generate effective test cases for randomly chosen targets in the subject benchmarks. Although each optimization algorithm performed well for generating target test data, when we compared the time efficiency, we found that ABC outperformed all other algorithms. Additionally, the fitness function designed using the proposed approach provided all the independent path coverage if

[3]Detailed MATLAB code for PSO for TCP can be provided by the authors upon prior request for academic use only.

all test requirements were considered while forming the objective fitness function.

A second major contribution of this work was demonstrated by solving the TCP problem using a PSO-based approach. We considered four real test pool sizes: one was small and the other three were large. For our experiments, for small and large test suites, we found that PSO-based ordering was substantially significant in maximizing the rate of fault detection in comparison with existing approaches, such as random, reverse, and unordered prioritization schemes.

A major problem that we found during this work was in the extraction of datasets for TCP. Public repositories are readily available; however, users extracting data from these available repositories encounter a small amount of user documentation. The preprocessing of raw data to make it useful requires a large amount of effort. Thus, care should be taken to provide user-friendly documentation for public datasets. The approach presented in this paper shows the potential of metaheuristic algorithms and techniques in the domains of ATCG and TCP. Thus, in future work, we will exploit existing nature-inspired algorithms and explore new paradigms of computational intelligence to solve various existing and forthcoming problems in software testing.

Appendix 1: Small application test suite- fault matrix

Only the main modules are provided in this paper. Detailed source code is available from the authors upon prior request.

Small- Size Application test Suite- Fault Matrix

Test case number	fault1	fault2	fault3	fault4	fault5
T1	X	–	–	–	–
T2	X	–	–	–	–
T3	X	X	–	–	–
T4	X	–	–	–	–
T5	X	–	–	–	–
T6	X	–	X	X	X
T7	X	X	–	–	–
T8	X	–	–	–	–
T9	X	–	–	–	–
T10	X	–	–	–	–
T11	X	–	–	–	–
T12	X	–	–	–	–
T13	X	–	–	–	–
T14	X	–	–	–	–
T15	X	–	–	–	–
T16	X	–	–	–	–
T17	X	–	–	–	–
T18	X	X	–	–	–

X- indicate fault is revealed and "–" indicates fault is not revealed

Large real application test suite: data can be accessed at [57].

Appendix 2: Objective function formulation for TeraPaint3

The following code is implemented in MATLAB and forms the objective function. The objective function is given as an input to the PSO algorithm. Note that the lower and upper bounds in the PSO algorithm depend on the system under test. Because *TeraPaint3* has 424 test cases, the lower and upper bounds are set to [1] and [424], respectively.

Objective Function Formulation for TeraPaint3

```
function [ObjVal test]=ofun_Terapaint(x)
T1= x(1);
T2= x(2);
T3= x(3);
T4= x(4);
.
T424= x(424);
  %F1...FN are the test cases:-Put here which test case
  find which faults(T)
F1= T39;
F2= T103;
F3= T113+T114+T115;
F4= T107+T108+T109;
F5= T110+T111+T112;
F6= T140;
F7= T51+T52+T53+T54
F8= T70+T73+T74+T75;
F9= T64+T67+T68+T69;
.
.
F420= T5+T6+T7+T8+T9+T10;
F421= T5+T6+T7+T8+T9+T10;
F422= T5+T6+T7+T8+T9+T10;
F423= T5+T6+T7+T8+T9+T10;
F424= T5+T6+T7+T8+T9+T10;
final= [F1 F2 F3 F4 .... F420 F421 F422 F423 F424];
[value, test] = sort(final, 2,'descend')
  ObjVal=(F1+F2+F3+F4..+ ...... +F420+F421+
  F422+F423+F424);
MY(:,:)=final;
End
```

% The APFD scores for the other two applications, *TeraPresent3* and *TeraSpreadSheet3,* are calculated in a similar manner.

References

1. Tassey G (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project 7007
2. Beizerm B (1990) Software Testing Techniques, Van Nostrand Reinhold. New York
3. Rothermel G, Untch RH, Chu C, Harrold MJ (1999) Test case prioritization: An empirical study. In: Proceedings IEEE international conference on software maintenance, (ICSM'99). IEEE, pp 179–188
4. Hou Y, Zhao C, Liao Y (2006) A new method of test generation for sequential circuits. In: International conference on communications, circuits and systems proceedings. IEEE, pp 2181–2185
5. Liang Y, Liu L, Wang D, Wu R (2010) Optimizing particle swarm optimization to solve knapsack problem Information computing and applications. Springer, pp 437–443
6. Liu H, Sun S, Abraham A (2006) Particle swarm approach to scheduling work-flow applications in distributed data-intensive computing environments. In: Sixth international conference on intelligent systems design and applications, ISDA'06, pp 661–666
7. Lope HS, Coelho LS (2005) Particle swarn optimization with fast local search for the blind traveling salesman problem. In: Fifth international conference on hybrid intelligent systems, HIS'05. IEEE, pp 245–250
8. Zhao F, Zhang Q, Yang Y (2006) An improved particle swarm optimization-based approach for production scheduling problems. In: IEEE international conference on mechatronics and automation. IEEE, 2279–2283
9. Do H, Rothermel G (2006) On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans Softw Eng 32:733–752
10. Elbaum S, Malishevsky AG, Rothermel G (2002) Test case prioritization: A family of empirical studies. IEEE Trans Softw Eng 28:159–182
11. Arcuri A, Briand L (2012) Formal analysis of the probability of interaction fault detection using random testing. IEEE Trans Softw Eng 38:1088–1099
12. Mills HD, Dyer M, Linger RC (1987) Cleanroom software engineering. IEEE Softw 4:19
13. Voas J, Morell L, Miller K (1991) Predicting where faults can hide from testing. IEEE Softw 8:41–48
14. Bird DL, Munoz CU (1983) Automatic generation of random self-checking test cases. IBM Syst J 22:229–245
15. Ferguson R, Korel B (1996) The chaining approach for software test data generation. ACM Trans Softw Eng Methodol (TOSEM) 5:63–86
16. Chang K-H, Cross IIJH, Carlisle WH, Brown DB (1992) A framework for intelligent test data generation. J Intell Robot Syst 5:147–165
17. Korel B (1996) Automated test data generation for programs with procedures ACM SIGSOFT software engineering notes. ACM, pp 209–215
18. Korel B (1990) A dynamic approach of test data generation. In: Proceedings, conference on software maintenance. IEEE, pp 311–317
19. McMinn P (2004) Search-based software test data generation: a survey. Software Testing Verification and Reliability 14:105–156
20. Clarke LA (1976) A system to generate test data and symbolically execute programs. IEEE Trans Softw Eng :215–222
21. Howden WE (1977) Symbolic testing and the DISSECT symbolic evaluation system. IEEE Trans Softw Eng :266–278
22. Ramamoorthy CV, Ho S-B, Chen WT (1976) On the automated generation of program test data. IEEE Trans Softw Eng :293–300
23. Shan J-H, Wang J, Qi Z-C (2004) Survey on path-wise automatic generation of test data. Acta Electron Sin 32:109–113
24. Malhotra R, Garg M (2011) An adequacy based test data generation technique using genetic algorithms. J Inf Process Syst 7:363–384
25. Michael CC, McGraw G, Schatz MA (2001) Generating software test data by evolution. IEEE Trans Softw Eng 27:1085–1110
26. Fraser G, Arcuri A (2014) A large-scale evaluation of automated unit test generation using EvoSuite. ACM Trans Softw Eng Methodol (TOSEM) 24:8
27. Tracey N, Clark J, Mander K (1998) Automated program flaw finding using simulated annealing ACM SIGSOFT software engineering notes. ACM, pp 73–81
28. Malhotra R, Khari M (2013) Heuristic search-based approach for automated test data generation: a survey. International Journal of Bio-Inspired Computation 5:1–18
29. Do H, Rothermel G, Kinneer A (2004) Empirical studies of test case prioritization in a JUnit testing environment. In: 5th International symposium on software reliability engineering ISSRE. IEEE, pp 113–124
30. Kim J-M, Porter A (2002) A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24rd international conference on software engineering, ICSE. IEEE, pp 119–129
31. Rothermel G, Untch RH, Chu C, Harrold MJ (2001) Prioritizing test cases for regression testing. IEEE Trans Softw Eng 27:929–948
32. Leon D, Podgurski A (2003) A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In: 14th international symposium on software reliability engineering, ISSRE. IEEE, pp 442–453
33. Mirarab S, Tahvildari L (2008) An empirical study on bayesian network-based approach for test case prioritization. In: 1st International conference on software testing, verification, and validation. IEEE, pp 278–287
34. Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Software testing, verification and reliability 22:67–120
35. Li Z, Harman M, Hierons RM (2007) Search algorithms for regression test case prioritization. IEEE Trans Softw Eng 33:225–237
36. Kaur DA, Goyal S (2011) A bee colony optimization algorithm for code coverage test suite prioritization. Int J Eng Sci Technol 1:2786–2795
37. Catal C, Mishra D (2013) Test case prioritization: a systematic mapping study. Softw Qual J 21:445–478
38. Jacob TP, Ravi T (2013) Optimization of test cases by prioritization. J Comput Sci 9:972
39. Singh Y, Kaur A, Suri B (2010) Test case prioritization using ant colony optimization. ACM SIGSOFT Software Engineering Notes 35:1–7
40. Ahmed AA, Shaheen M, Kosba E (2012) Software testing suite prioritization using multi-criteria fitness function. In: 222nd International conference on computer theory and applications (ICCTA). IEEE, pp 160–166
41. Chen Y, Zhong Y (2008) Automatic path-oriented test data generation using a multi-population genetic algorithm. In: Fourth international conference on natural computation, ICNC'08. IEEE, pp 566–570
42. Mohapatra D (2011) GA Based Test Case Generation Approach for Formation of Efficient Set of Dynamic Slices. International Journal on Computer Science and Engineering (IJCSE) 3:
43. Pargas RP, Harrold MJ, Peck RR (1999) Test-data generation using genetic algorithms. Software Testing Verification and Reliability 9:263–282

44. Berndt D, Fisher J, Johnson L et al (2003) Breeding software test cases with genetic algorithms. In: Proceedings of the 36th annual hawaii international conference on system sciences
45. Ahmed MA, Hermadi I (2008) GA-Based multiple paths test data generator. Comput Oper Res 35:3107–3124
46. Mukesh MOPS (2015) Generating and prioritizing optimal paths using ant colony optimization. Computational Ecology and Software 5:1
47. Watkins AL (1995) The automatic generation of test data using genetic algorithms. In: Proceedings of the 4th software quality conference, pp 300–309
48. Flipcart Shop Flipcart, availabte: www.flipcart.com. Accessed June 2016
49. Arcuri A, Briand L (2014) A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability 24:219–250
50. Rothermel G, Harrold MJ (1996) Analyzing regression test selection techniques. IEEE Trans Softw Eng 22:529–551
51. Di Caro G, Dorigo M (1998) Antnet: Distributed stigmergetic control for communications networks. J Artif Intell Res :317–365
52. Dorigo M, Maniezzo V, Colorni A (1996) Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics 26:29–41
53. Li H, Lam CP (2004) Software Test Data Generation using Ant Colony Optimization. In: International conference on computational intelligence, pp 1–4
54. Ayari K, Bouktif S, Antoniol G (2007) Automatic mutation test input data generation via ant colony. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation. ACM, pp 1074–1081
55. Parpinelli RS, Lopes HS, Freitas AA (2002) Data mining with an ant colony optimization algorithm. IEEE Trans Evol Comput 6:321–332
56. Zhao P, Zhao P, Zhang X (2006) A new ant colony optimization for the knapsack problem. In: CAIDCD'06 7th international conference on computer-aided industrial design and conceptual design. IEEE, pp 1–3
57. Atif M (2016) Software Benchmark repository for TeraPaint3. http://www.cs.umd.edu/~atif/Benchmarks/common/TerpPaint3-fault_matrix.txt. Accessed 1 June 2016
58. Atif M (2016) Software Benchmark repository for TeraPresent3. http://www.cs.umd.edu/~atif/Benchmarks/common/TerpPresent3-fault_matrix.txt. Accessed 20 June 2016
59. Atif M (2016) Software Benchmark repository for TerpSpreadSheet3, available. http://www.cs.umd.edu/~atif/Benchmarks/common/TerpSpreadSheet3-fault_matrix.txt. Accessed 20 June 2016
60. Krishnamoorthi R, Mary SASA (2009) Regression test suite prioritization using genetic algorithms. International Journal of Hybrid Information Technology 2:35–52
61. Srivastava PR (2008) Test case prioritization. Journal of Theoretical and Applied Information Technology 4:178–181
62. Kaur A, Bhatt D (2011) Hybrid particle swarm optimization for regression testing. Int J Comput Sci Eng 3:1815–1824
63. Kaur A, Bhatt D (2011) Particle swarm optimization with crossover operator for prioritization in regression testing. Int J Comput Appl 27:27–34

Mr. Mukesh Mann is a researcher in the Department of Computer Science and Engineering, School of ICT, Gautam Buddha University (GBU), Greater Noida. In past, he has received his M.Tech in Information and Communication Technology from GBU, India and B.Tech in Computer Science and Engineering from Kurukshetra University, Haryana, India.

He had received Direct Senior Research Fellow award in Computer Science and Engineering by Council of Scientific and Industrial Research, Govt. of India in 2014.He is also awarded Junior Research fellowship award- 2014 and Senior Research Fellow award-2016 by University Grant Commission (UGC), Govt. of India for pursuing Ph.D. in Computer Science and Engineering. His area of research is Software Engineering, Computational Intelligence and Machine Learning.

Dr. Pradeep Tomar received his Ph.D. in computer science and engineering from M. D. University, Rohtak, Haryana, India. His area of research is Computational intelligence and Component-based Software Engineering. He is a life member of Computer Society of India. Presently he is working as an Assistant Professor in the School of Information and Communication Technology, Gautam Buddha University, Greater Noida, U.P., India.

Dr. Om Prakash Sangwan received his Ph.D. and M.Tech in Computer Science & Engineering from Guru Jambheshwar University of Science & Technology, Hisar, Haryana, India. His area of research is Software Engineering and Soft Computing. He is a life member of Computer Society of India. Presently he is working as an Associate Professor in Guru Jambheshwar University of Science and Technology, Hisar, India.